

Universidade do Minho
Escola de Engenharia

Rui Sérgio Magalhães da Costa

Scalar algorithms for molecular docking in heterogeneous platforms

Tese de Mestrado
Mestrado de Engenharia Informática

Trabalho efectuado sob a orientação de
Alberto Proença
Nuno Micaêlo

ABSTRACT

The high throughput screening of new candidate drugs uses computational intensive molecular docking simulations. State-of-the art implementations for multicore-CPU systems still have performance, precision and accuracy limitations, which require an increase in the efficiency and scalability of both molecular docking algorithms and their coding. Current heterogeneous platforms that merge multicore CPU with CUDA enabled GPU devices are an affordable accelerating technology that may overcome the performance limitations.

The dissertation work aims to efficiently port to an heterogeneous platform a popular open source software package for molecular docking, Autodock Vina, keeping the functionality of the original algorithms whenever feasible. The new version, ScalaVina, predicts the noncovalent chemical interaction of a small molecule (ligand) against the binding site of a receptor macromolecule (receptor), evaluating the fitness of the ligand inside the binding pocket using a scoring function, and searching the best structural fit between ligand and receptor with the lowest local minima binding energy.

The original Vina supports multithreaded parallelism at a high level, by launching multiple starting points with sequential activities, which include the global optimizer heuristic algorithm iterated local search, ILS, the energy minimization function using the BFGS algorithm, the multi-variable scoring function and other support functions. The development of the ScalaVina version to take advantage of a CUDA enabled GPU required the parallelization of these functions under the SIMD paradigm, while attempting to minimize data transfers to/from the accelerating board and to improve data access patterns. Each starting point was mapped into one streaming multiprocessor at the GPU device, exploring the SIMD multithreaded parallelism to compute the values required for the docking operations above mentioned. This approach mimics the original Vina when it allocates a hardware block (SM) in the GPU for each starting point; however, these functions are executed in parallel in the GPU.

Obtained results show that the GPU output of ScalaVina is in qualitative and quantitative agreement with experimental data and closely matches the output of Vina for the same set of receptor:ligand pairs. Preliminary performance results also show that there is room for improvements with careful tuning.

Keywords: GPU computing, molecular docking, heterogeneous computing

RESUMO

A triagem de grande volume de novos candidatos para medicamentos usa simulações de *docking* molecular computacionalmente intensivas. O estado da arte em implementações para sistemas CPU *multicore* ainda têm limitações de desempenho, precisão e exactidão, que requerem uma melhoria na eficiência e escalabilidade de ambos os algoritmos de *docking* molecular e a sua codificação. Sistemas heterogêneos que combinam CPUs *multicore* e GPUs com CUDA são uma tecnologia de aceleração acessível que poderá superar as limitações de desempenho.

Esta dissertação tem como objectivo transpor para uma plataforma heterogênea um pacote de *software open source* de *docking* molecular popular, Autodock Vina, mantendo a funcionalidade dos algoritmos originais onde possível. A nova versão, ScalaVina, prediz a interacção química não-covalente de uma pequena molécula (ligando) contra o centro activo de uma macromolécula receptora (receptor), avaliando a aptidão do ligando dentro do centro activo utilizando uma função de *scoring*, e procurando o melhor encaixe estrutural entre o ligando e o receptor com a energia de ligação mais baixa.

O Vina original suporta paralelismo via *multithreading* a um alto nível, lançando várias actividades sequenciais cada uma a partir de um ponto inicial diferente, actividades as quais incluem um algoritmo heurístico para a optimização global *iterated local search*, ILS, a função de minimização de energia utilizando o algoritmo BFGS, a função de *scoring* multi-variável e outras funções de suporte. O desenvolvimento da versão ScalaVina para tirar partido de um GPU com CUDA necessitou da paralelização dessas funções dentro do paradigma SIMD, enquanto minimizando transferências de dados de/para a placa de aceleração e melhorar os padrões de acesso aos dados.

Cada ponto inicial foi mapeado para um *streaming* multiprocessor no dispositivo GPU, explorando o paralelismo SIMD para calcular os valores requeridos para as operações de *docking* acima mencionadas. Esta aproximação imita o Vina original em que para cada bloco de *hardware* (SM) é alocado um ponto inicial; no entanto, estas funções são executadas em paralelo no GPU.

Resultados obtidos mostram que os dados de saída em GPU do ScalaVina estão qualitativamente e quantitativamente em concordância com os dados experimentais e se aproxima dos dados de saída do Vina para o mesmo grupo de pares receptor:ligando. Resultados preliminares de desempenho mostram também que há espaço para melhoramentos com afinamentos cuidadosos.

Keywords: GPU computing, molecular docking, heterogeneous computing

ACKNOWLEDGMENTS

It was a long road getting to this point and I could not have done it without the support I've had. I thank my supervisors first of all, Prof. Alberto Proença and Doutor Nuno Micaêlo, for everything they have done. I also thank Prof. Rui Ralha for his assistance when I asked for his help. I thank my family for putting up with my occasional sleepless night and my friends for helping me relax.

I also acknowledge the research scholarship awarded by Universidade of Minho that contributed to a smoother work realization.

I would thank my cats, too, but they do not care about what I am doing if they decide they want a warm lap, no matter how inconvenient... cute furballs.

Contents

Table of Contents	ix
1 Introduction	1
1.1 Context	1
1.2 Goals	2
1.3 Dissertation Structure	3
2 Molecular docking	5
2.1 Scoring function	6
2.2 Optimization algorithms	11
3 Porting the docking algorithm for CUDA-enabled devices	13
3.1 CUDA model	13
3.2 Mapping algorithms and data structures into an heterogeneous platform	17
3.3 Scoring	21
3.4 Search for global minimum	28
3.5 Search for local minima	29
4 Validation and Assessment	35
4.1 Functional validation	35
4.2 Testbed platform	37
4.3 Performance evaluation	40
5 Conclusions	45
Bibliography	49

Chapter 1

Introduction

1.1 Context

The rise of the computer as a scientific research tool is an apparent and pervasive fact in modern society which has greatly influenced many scientific fields and the processes associated with them. One such process where computers are making significant improvements is on the drug development process. [1]

One of the initial phases of drug development is the experimental screening of hundreds, if not thousands, of molecules using high throughput *in vitro* biological assays and medium throughput animal assays. The best candidates are then selected for pre-clinical testing and then onwards. [2,3]

The main problem with screening is that the sheer number of molecules that exist, let alone those that could theoretically be synthesized, makes it impractical to screen all of them through experimental means, not to mention how attempting such would be hideously expensive. How, then, can millions of molecules be sifted through and screened to look for the best subset of candidates for chemical synthesis and biological testing? One of the current approaches is through molecular docking simulation.

Molecular docking is used to screen large libraries of candidate molecules and filter all but the best N candidate molecules (being N an arbitrary amount) for further experimental studies. This enables researchers to screen an otherwise impossible range of different molecules at a fraction of the time and price it would otherwise cost, thus contributing to both increase drug quality and decrease cost. [4–6]

Molecular docking itself is a problem of optimization where the computer tries to find, per molecule, the best fit of a candidate drug when attempting to bind against, typically, a protein. The molecule, also referred to as ligand from now on, is typically subjected to translations, rotations and torsional conformation changes. At each new conformation a scoring function is computed to assess how good the solution is. The accuracy of each solution depends on the parameters, algorithms and scoring function (or functions) used and, normally, better accuracy comes at the cost of slower calculations, affecting overall throughput.

Computer technology is in constant evolution and recent advances have made available to computer scientists the use of hardware that has previously been associated only with entertainment software. Graphics cards and its dedicated graphics processor, also known as GPU, are capable of massive parallel math computations on a chip, making them an affordable and ideal platform for applications that are numerically intensive.

The move towards heterogeneous computing platforms based on both multicore CPU and manycore GPU has already proved its success, where 3 out of the 5 top computers in the world in the June 2011 TOP500 list were HPC heterogeneous clusters.

1.2 Goals

The goals of this dissertation are to:

- Select and analyse an existing molecular docking software package developed for an homo-

geneous CPU-based platform, preferably open source and widely accepted by the scientific community

- Port the molecular docking algorithms and functionalities to a GPU device as appropriate ; not all algorithms are adequate for GPUs
- Assess the performance advantage of an efficient use of both multicore CPU and GPU performance features
- Compare the proposed solution with original CPU-only version and experimental results - validating the output results, as well as comparing performance both in terms of speed and quality.

1.3 Dissertation Structure

Here is a summary of the following chapters:

- Chapter 2 - Molecular docking: present the chosen software package, AutoDock Vina, and how it works. Includes technical information as needed to explain the inner workings.
- Chapter 3 - Porting the docking algorithm for CUDA-enabled devices: present ScalaVina's GPU pipeline, developed to behave like AutoDock Vina's docking algorithm, and explain how it works.
- Chapter 4 - Validation and Assessment: present an evaluation of the results obtained so far with the ScalaVina prototype.

Chapter 2

Molecular docking

The process of molecular docking attempts to find the best fit for a molecule (known as ligand) to bind against a specific site of a large biomolecule such as protein, enzyme or DNA, in the shortest amount of time possible. It is a problem that suffers from combinatorial explosion due to the large numbers of degrees of freedom and the vast conformational space accessible in this type of system, making efficient approaches and algorithms that attempt to deal with it effectively a necessity. One docking software package that has an efficient strategy is AutoDock Vina, combining speed with good accuracy and scalability with many-core CPUs.

AutoDock Vina uses a global heuristic optimizer algorithm, Iterated Local Search [7], with a local optimization algorithm called Broyed-Fletcher-Goldfarb-Shanno (BFGS) [8, 9] to find the best fits according to a scoring function that attempts to predict the binding affinity between the ligand and the receptor.

2.1 Scoring function

The general form of the conformation-dependent part of the scoring function used in Autodock Vina [5] is

$$c = \sum_{i < j} f_{t_i t_j}(r_{ij}) \quad (2.1)$$

where the summation is over all pairs of atoms that can move relative to each other, excluding 1-4 interactions (neighbour atoms separated by 3 chemical bonds). Each atom i is assigned a type t_i , and a symmetric set of interaction functions $f_{t_i t_j}$ of the interatomic distance r_{ij} should be defined. This value can be seen as a sum of intermolecular and intramolecular contributions:

$$c = c_{inter} + c_{intra} \quad (2.2)$$

The optimization algorithms, described below in their respective chapters, attempt to find the global minimum of c and other low-scoring conformations, which it then ranks according to the free energy of binding predicted from the intermolecular component of the lowest-scoring conformation, designated as 1:

$$s_1 = g(c_1 - c_{intra1}) = g(c_{inter1}) \quad (2.3)$$

where the function g can be an arbitrarily strictly increasing smooth function, not necessarily linear. In the output, other low-ranking conformation of interest are also assigned s values but, to preserve ranking, use c_{intra} from s_1 .

$$s_i = g(c_i - c_{intra1}) \quad (2.4)$$

Autodock Vina uses a modular design where the program does not rely on any particular form of $f_{t_i t_j}$, instead using an object instance passed as parameter as a functor to provide the required scoring function functionality. In addition, multiple and/or alternative atom typing schemes can be used, such as the AutoDock 4 atom typing [10] or SYBIL-based atom types [11].

Term	Weight
<i>gauss</i> ₁	-0.0356
<i>gauss</i> ₂	-0.00516
repulsion	0.840
hydrophobic	-0.0351
hydrogen bonding	-0.587
$g(c_{inter})$	0.0585

Table 2.1 Scoring function terms and weights

The scoring function implemented in AutoDock Vina is inspired by X-Score [12] and was tuned using the PDBbind [3]. Some of the terms are different, however, and tuning went beyond linear regression. Also, X-Score ranks conformations using different criteria, relying only on the intermolecular contribution. The atom typing scheme follows that of X-Score. The hydrogen atoms are not considered explicitly, other than for atom typing, and are omitted from 2.1. The derivation of AutoDock Vina’s scoring function combines knowledge-based potentials and empirical scoring functions.

The interaction functions f_{itj} , defined relative to the surface distance of the atoms ($d_{ij} = r_{ij} - R_{t_i} - R_{t_j}$ [13]), also referred to as the terms of the scoring function, are defined as:

$$f_{itj}(r_{ij}) \equiv h_{itj}(d_{ij}) \quad (2.5)$$

where R_t is the van der Waals radius of atom type t . In AutoDock Vina’s scoring function, h_{itj} is a weighted sum of three steric interactions, an hydrophobic interaction between hydrophobic atoms and, where applicable, hydrogen bonding interaction. Scoring function terms and weights are on figure 2.1.

The equations defining each term are as follows:

$$gauss_1(d) = e^{-(d/0.5\text{\AA})^2} \quad (2.6)$$

$$gauss_2(d) = e^{-((d-3\text{\AA})/2\text{\AA})^2} \quad (2.7)$$

$$repulsion(d) = \begin{cases} d^2, & \text{if } d < 0\text{\AA} \\ 0, & \text{if } d \geq 0\text{\AA} \end{cases} \quad (2.8)$$

$$hydrophobic(d) = \begin{cases} 1, & \text{if } d < 0.5\text{\AA} \\ 1.5 - d, & \text{if } 0.5\text{\AA} \leq d \leq 1.5\text{\AA} \\ 0, & \text{if } d > 1.5\text{\AA} \end{cases} \quad (2.9)$$

$$Hbond(d) = \begin{cases} 1, & \text{if } d < -0.7\text{\AA} \\ 1 - \frac{d+0.7}{0.7}, & \text{if } -0.7\text{\AA} \leq d \leq 0\text{\AA} \\ 0, & \text{if } d > 0\text{\AA} \end{cases} \quad (2.10)$$

For optimization reasons, the scoring function is discretized and pre-calculated into a series of look-up tables, each table holding the interactions between two atom types according to distance. Using simple interpolation, the values between the discretized points can be trivially computed without needing to execute the scoring function itself, which would take much longer on a CPU. In addition, using a pre-calculated table also makes computing the derivative trivially easy, storing in a look-up table as well. This allows a smoothing operation to be run on the scoring function's pre-calculated values to improve the line search performance, an algorithm later explained in section 2.2, Optimization algorithms.

Finally, the conformation-independent function g is defined as

$$g(c_{inter}) = \frac{c_{inter}}{1 + wN_{rot}} \quad (2.11)$$

where N_{rot} is the number of active rotatable bonds between heavy atoms in the ligand and w is

the associated weight.

Intermolecular ligand-receptor scoring

Determining the fitness of the ligand's conformation against all atoms of the receptor would be an excessively time consuming operation if the naive approach towards computing it was followed. Such approach would require computing the score of each of the ligand's atoms against every atom in the receptor. Considering the size of receptors is in the range of thousands of atoms larger, the cost would be prohibitive.

Instead, as most docking programs do, Autodock Vina computes 3D grid maps containing the volume of interest where the binding site is located, which it then uses in a similar way to the pre-calculated scoring function. Each point in each grid contains the score of an atom of type t if it were placed at exactly that location. Using interpolation, the scoring behaviour of the volume containing the binding site can be reasonably approximated, derivatives included. In order to nudge ligand atoms that may stray outside the volume, a penalty is awarded that is proportional to how far outside the grid the atom is, therefore guaranteeing that, eventually, all atoms will remain within the binding site volume. The derivatives are computed as a vector applied to the forces vector of each atom, which is later used to compute the force and torque of the ligand.

Generating the 3D grid maps is an easy task, though a single-threaded one, essentially iterating over the receptor atoms and computing their score against each grid point. As an obvious optimization, only the grid maps corresponding to atom types in the ligand are actually generated.

Intramolecular ligand scoring

Intramolecular scoring in AutoDock Vina is based on a list of interacting pairs, pre-computed when the ligand is processed, to exclude 1-4 interactions. Each element defines that a pair of atoms i and j interact with each other and need to be computed. This list of interactions is always

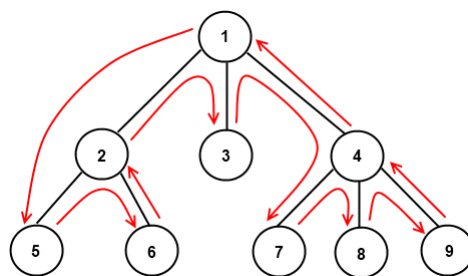


Figure 2.1 Depth-first, post-order tree traversal example. Root node is not visited until the end

much higher (except in trivial cases) than the number of atoms in the ligand and corresponds to $f_{t_i t_j}$ in the scoring function definition (see eq.2.1). The interactions are traversed in serial, computing the score of each in turn, accumulating the result.

Derivatives are, when necessary, computed at the same time. A force vector is computed and applied to each of the involved atoms' forces vector, which is later used to compute the ligand's force and torque.

Ligand force, torque and conformation update

The computed derivatives have a simple mechanical interpretation. With respect to the position, orientation and torsions, the derivatives are the negative total force acting on the ligand, the negative total torque and the negative torque projections respectively, where the projections refer to the torque applied to the branch moved by the torsion, projected on its rotation axis. [5]

AutoDock Vina computes the force and torque by recursively traversing a tree-like data structure representing the ligand's structure depth-first and accumulating the forces on one vector and the torque in another. This requires that, on each tree node, the individual atoms associated with it have their forces accumulated, as well as the computed torque. This is all sequentially performed in postorder (see figure 2.1), starting at the end of the branch(es) and computing upwards.

Updating a conformation is a similar process to computing the force and torque, except the

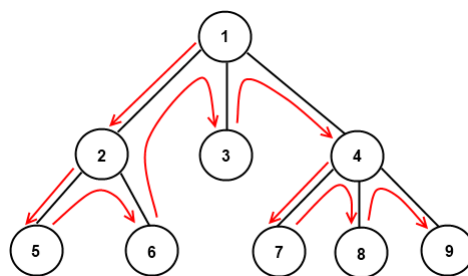


Figure 2.2 Depth-first, pre-order tree traversal example. Root node is visited at the start

traversal is preorder (see figure 2.2), computing from the root node onwards. At each node the position, orientation and axis are all updated, then the atom positions are computed.

2.2 Optimization algorithms

AutoDock Vina uses a combination of algorithms to perform the binding site exploration to search for the best fit. At the highest level a heuristic method called Iterated Local Search [8,9] is used to explore the search area as much as possible. The BFGS algorithm, an efficient quasi-Newton method, is used to find the local minima (energy minimization) from each starting conformation point. The minima are then evaluated and either accepted or rejected by the ILS algorithm according to the Metropolis criterion. If accepted, the result is further refined before inclusion in the output set for the ILS run in question and, after perturbing the conformation found, runs another energy minimization and so on, until the ILS algorithm halts.

AutoDock Vina has the capability of launching multiple threads, each with its own ILS run from a different starting point. The ILS algorithm, and everything else that runs inside, is executed sequentially. It executes the same algorithm from different starting points as an effective way of implementing parallelism, especially as there is no shared data among concurrent threads. Keeping the processes isolated also has the advantage of dismissing most parallel execution issues, such as synchronization, load balancing, bugs and others.

An important note here is that the BFGS algorithm includes a line search algorithm, responsible for determining how far along the scoring function gradient to travel. The quicker the line search algorithm finds a good distance, the faster the docking overall will be. This is obvious, but since every time a line search iteration is performed there is also a conformation update and a scoring function call, so the faster it can be made to converge, the less frequently the scoring function has to be called. Part of the optimizations done in AutoDock Vina was precisely line search convergence optimizations, namely smoothing the scoring function.

Chapter 3

Porting the docking algorithm for CUDA-enabled devices

3.1 CUDA model

CUDA is a technology developed by NVIDIA designed to give developers the capability to use GPUs as general computation accelerator platforms. The CUDA execution model consists of a *Host*, which is a traditional CPU such as Intel or AMD processors, and one or more *Devices*, which are the CUDA-enabled massively parallel processors installed on the host. These CUDA-enabled devices are available solely from NVIDIA currently and take the form of either consumer-grade graphics cards like the GeForce family or products or professional grade accelerator cards such as the Tesla family. [14]

Fermi

The Fermi architecture is the latest CUDA-enabled GPU architecture developed by NVIDIA. Among its key features are support for 14 to 16 Streaming Processors (SM) each with 32 CUDA

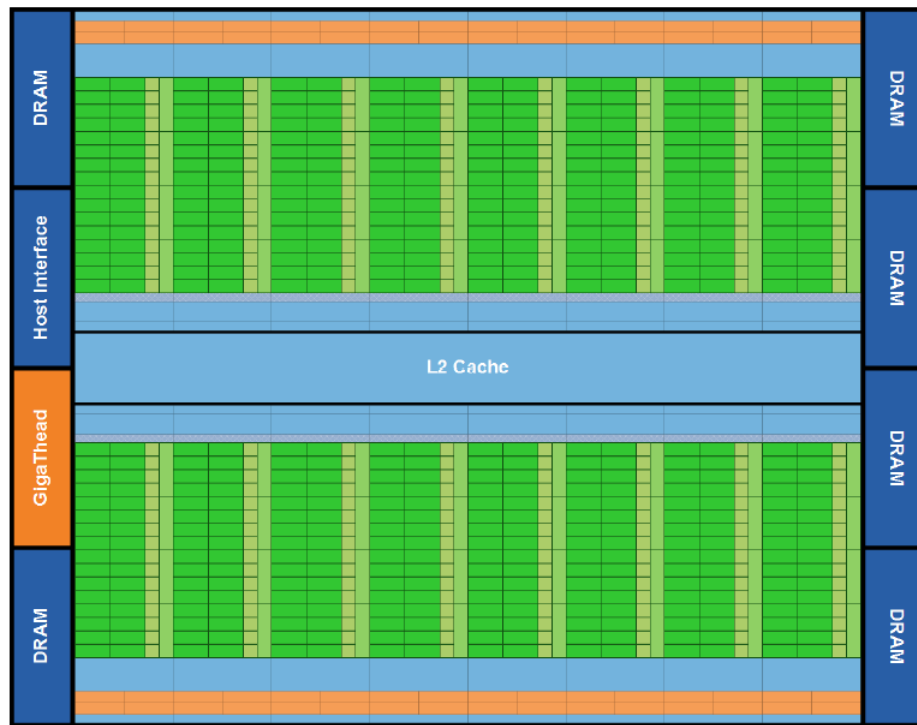


Figure 3.1 Fermi architecture overview [15]

cores, 64kB RAM per SM dedicated for cache and shared memory, full C++ support, full single and double precision FP support and ECC memory. [15]

Fermi-based accelerator cards, as all GPUs, are built to be massively parallel floating point computing devices, due to the prevalence of such operations in modern entertainment software. Unlike CPUs which dedicate large parts of the die surface to memory and are highly optimized towards sequential workloads (even as more CPU cores are added), GPUs dedicate the vast majority of their die surface towards computing cores and have been optimized for throughput. Due to fundamentally weaker memory model constraints, GPUs are also able to reach much higher memory bandwidths than traditional CPUs, further compounding the advantage in parallel workloads. [14]

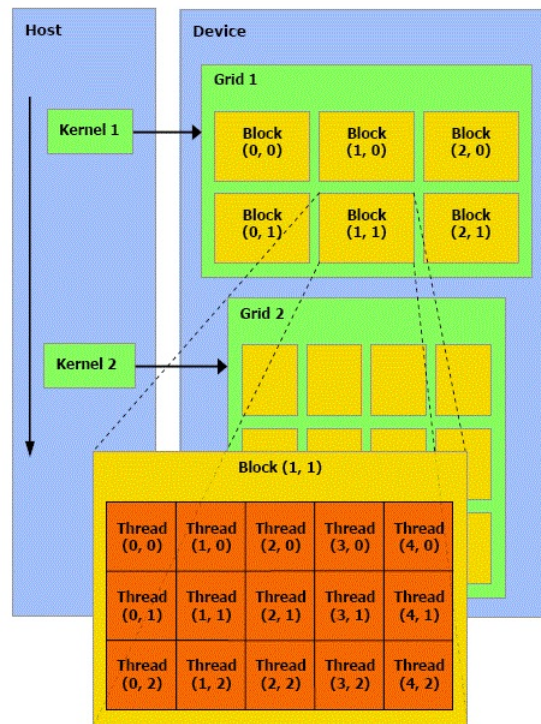


Figure 3.2 CUDA thread organization [14]

CUDA programming model

The CUDA programming model is designed for many-core architectures with wide SIMD parallelism, such as the Fermi. In this model, the host is responsible for setting up the kernel, which is the device-side part of the CUDA program, launch it and then retrieving the results when it finishes. Structurally, each kernel has one grid composed of blocks of threads. Each block is concurrently and independently executed of all other blocks as determined by the device depending on resource constraints. Cooperation between different blocks is not supported, as synchronization is done only through shared memory at the block level. Threads are grouped into warps, which execute in lockstep in the SIMD paradigm.

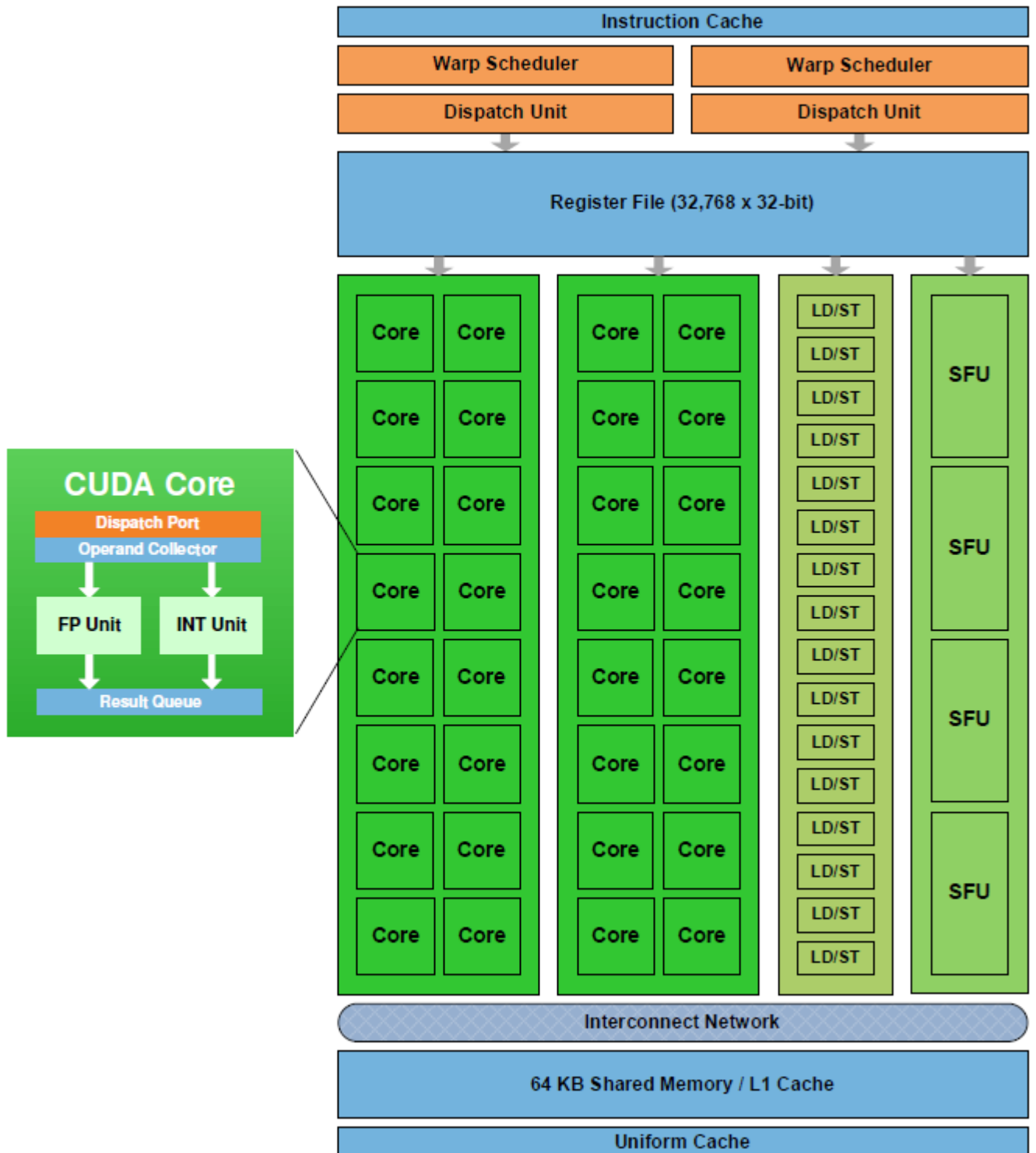


Figure 3.3 Fermi Streaming Multiprocessor [15]

3.2 Mapping algorithms and data structures into an heterogeneous platform

The process of molecular docking currently in ScalaVina is implemented through a pipeline that inputs data structures from Vina's parser and grid generator, issues multiple computing processes for the simulation and formats results to export to Vina to filter and store the relevant conformations. The computing processes are an abstraction to hide the hardware behind the activities, that could be a set of CPU cores or other accelerator component, such as a GPU, an FPGA or a DSP, available locally or remotely.

A CUDA-based computing process enables usage of CUDA-enabled devices (e.g. a GPU) for molecular docking and converts the CPU-centric Vina data structures into CUDA-optimized and compliant data structures for processing and then retrieve and process the output for storage and subsequent use by the user.

An overview of the entire docking process showed that the overhead of passing data back and forth for the CPU to make decisions would become both onerous to optimize and a potential bottleneck. As a consequence, the pipeline design inside the CUDA computing process was primarily optimized for minimal data movement between CPU and GPU. A second optimization aimed to take advantage of as much parallelism as possible for the common case. This led to the current design, based on one block per starting point, eliminating any need for kernel-wide synchronization and optimizing thread-usage. These design choices made the kernel both much more resilient and more predictable, as inter-block synchronization is a feature that is not officially supported by any CUDA-enabled device (though it is technically possible, albeit costly).

Data structures

The docking process relies on a host of data structures to store both read-only data as well as read/write and write only data structures. In some cases it was not readily apparent on the initial design which data fields fell into each category, so some potentially sub-optimal choices were made early on, especially as there is such a sheer breadth of different data structures.

Following comes a list of relevant top-level data structures and their purpose. Not all structures are listed.

- **Model** - the Model is a container structure encapsulating all the required data to run a docking simulation, including receptor and ligand data.
- **Scoring function** - the Scoring function is a functor-type structure that provides functionality to evaluate a ligand's conformation against a set of criteria.
- **Grid** - the Grid contains a 3D spatial map of the energy distribution of the receptor as pre-calculated using a given Scoring Function.
- **Conformation** - the Conformation structure contains the positional, rotational and angular information required to set a new conformation from the Model's original conformation.

All these data structures have both a CPU and a CUDA version. Listed are only those types that are used for the entire pipeline. For some specialist uses, other data types may be used in places, such as when calculating only the score of a model where it makes sense to skip using a pre-calculated Grid.

Under the hood, the Model structure is by far the most complex, with 3 positional coordinate vectors, two sets of atomic information (one for receptor, one for ligands) and then structural information for the ligands. Note, however, that the CUDA implementation currently only supports a single ligand. Supporting multiple ligands requires considerable changes in some algorithms and

Data Type	CPU name	CUDA name
Model	model	cudaModel
Scoring function	weighted_terms	cudaStandardWeightedTerms
Grid	cache	cudaSimpleCachedGridHolder
Conformation	conf	cudaConformationState

Table 3.1 Data structures, naming and mapping

due to the required additional complexity, it was dropped in favour of a simpler implementation that could run faster and be easier to validate the results. From that point, multiple ligand, flexible side-chains and residue support could be built later, backed by first-hand experience. This approach may have turned out to be critical to success, as the complexity of the entire process was not apparent at earlier stages.

The data structures encapsulated by the Model structure were redesigned in the CUDA implementation in order to improve parallel data access patterns. To do so, data was organized in patterns of *structure of arrays*, as an alternative to the original *array of structures*. This modification allows accessing the same variable from a contiguous sequence of elements in the array (such as, for example, atoms) with a single memory load operation while maximizing efficiency by not loading unnecessary data. None of the data structures required additional transformations except the ligand's structural data which, in AutoDock Vina, exists in the form of a tree based on pointers and recursivity for navigation.

The ligand structure in ScalaVina was remapped as an array in which each element corresponds to one of the nodes of the tree, like in figure 3.4. Each element records information such as the depth, index of the parent tree node and other values required for navigation, in addition to the structural data associated with the ligand's branches. An algorithm, later described, is used to traverse in parallel the tree instead of relying on a sequential recursive algorithm to do it. To

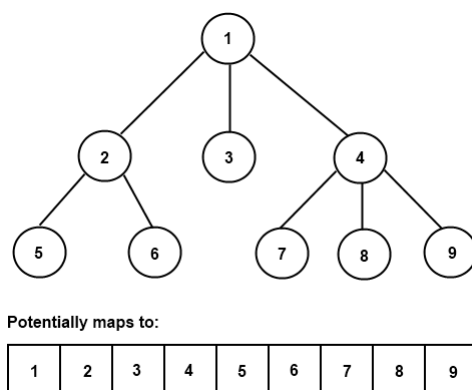


Figure 3.4 Mapping the structure tree into a vector for parallel, iterative processing

facilitate conformation updates in CUDA, a frame index was added to each atomic information entry, which contains the index of the branch that atom belongs to. This enables a fully parallel operation on the atoms to access structural information without having to actually navigate through the structure, a feature also exploited during conformation updates.

The Grid structure suffered no significant changes in the CUDA implementation other than recoding for compatibility and merits no specific discussion. An alternative Grid implementation that would benefit from an improved memory access pattern is technically possible but would cost roughly eight times more memory and be more complex to implement in practice. This is discussed in section 3.3 .

To the CUDA-based Model structure, a hint table was added recently to store information about what order the force vectors of the atoms may be updated in to avoid concurrent conflicting updates. This is composed of n rows of interacting pair indices (or a value that means nothing should be done), built at the same time the CPU-based model is converted to the CUDA-based model.

Data organization

The CUDA implementation in ScalaVina is designed to take the most advantage possible of cache memory first and use shared memory for every operation it is feasible to use it for to execute faster without imposing limitations. For this matter, the reference kernel does not store any atomic information on shared memory as doing so will automatically impose limitations on what kind of simulations can be run. Keeping shared memory use low enough to make use of 48kB of cache was the plan to keep performance high with the reference docking code, a plan that runs into difficulties with the inability to actually enforce how many blocks can run concurrently on a single SM. The consequences of this inability are explored in chapter 4.

3.3 Scoring

Scoring function

The scoring function used in ScalaVina uses the same methodology as Autodock Vina [5], although implementation details differ. The general form of the conformation-dependent part of the scoring function is the same as explained in chapter 2 and the conformation-independent part relies on AutoDock Vina's code.

$$c = \sum_{i < j} f_{itj}(r_{ij}) \quad (3.1)$$

$$c = c_{inter} + c_{intra} \quad (3.2)$$

The optimization algorithms, described in their respective sections, attempt to find the global minimum of c and other low-scoring conformations, which it then ranks using exactly the same functions as Autodock Vina does. Predicting the free energy of binding works the same way as in Autodock Vina.

Term	Weight	CPU Impl.	CUDA Impl.
$gauss_1$	-0.0356	✓	✓
$gauss_2$	-0.00516	✓	✓
repulsion	0.840	✓	✓
hydrophobic	-0.0351	✓	✓
hydrogen bonding	-0.587	✓	✓
$g(c_{inter})$	0.0585	✓	✗

Table 3.2 Scoring function weights, terms and available implementations

ScalaVina preserves modularity through the usage of C++ template techniques, therefore allowing compile-time swapping of scoring functions, and it also allows alternative atom typing schemes to be used. These design considerations allow major functionality changes while limiting the amount of required coding.

The interaction functions f_{t,t_j} , are defined the same way as AutoDock Vina's. Note, however, that N_{rot} is not implemented in CUDA on purpose since it is independent of the conformation of the ligand and not used when searching for the best fit. The terms, weights and implemented platforms are on table 3.2.

Definitions for the derivatives of the terms are also required in the CUDA implementation. This is due to the fact that, while in AutoDock Vina the scoring function is pre-calculated into look-up tables (and discretized in the process) and the derivatives implicitly obtained from there, the ScalaVina scoring function in CUDA computes the actual derivative without accessing a look-up table. This change has implications only when computing intramolecular energies and intermolecular energies where the receptor is not involved (such as residue and flexible side-chains), but is a relevant difference. On first glance, it may seem that a look-up table would be a superior choice but raw floating point computing power is a strength, not a weakness, of CUDA-enabled GPUs.

Furthermore, the memory access pattern would be effectively random and impossible to predict, since the table per thread to look-up the required value depends on both atom types. The certain outcome of this would be an extremely inefficient memory access pattern that nullifies any possible advantage over not computing the values, and in addition it poisons the precious little amount of available cache memory. CPUs may also suffer from the completely random access pattern, but modern CPUs have significantly larger cache memories, thus being able to keep all data much closer to the computing unit than GPUs are able to.

There is, however, no technical reason preventing an implementation of a look-up table based scoring function class like AutoDock Vina uses.

The equations defining each derivative are defined as follows:

$$gauss_1(d)' = \frac{2de^{-\frac{d^2}{0.5^2}\text{\AA}}}{0.5^2\text{\AA}} \quad (3.3)$$

$$gauss_2(d)' = \frac{2(d-3)e^{-\frac{(d-3)^2}{4}\text{\AA}}}{4\text{\AA}} \quad (3.4)$$

$$repulsion(d)' = \begin{cases} 2d, & \text{if } d < 0\text{\AA} \\ 0, & \text{if } d \geq 0\text{\AA} \end{cases} \quad (3.5)$$

$$hydrophobic(d)' = \begin{cases} 0, & \text{if } d < 0.5\text{\AA} \\ -1, & \text{if } 0.5\text{\AA} \leq d \leq 1.5\text{\AA} \\ 0\text{\AA}, & \text{if } d > 1.5\text{\AA} \end{cases} \quad (3.6)$$

$$Hbond(d)' = \begin{cases} 0, & \text{if } d < -0.7\text{\AA} \\ -\frac{1}{0.7}, & \text{if } -0.7\text{\AA} \leq d \leq 0\text{\AA} \\ 0, & \text{if } d > 0\text{\AA} \end{cases} \quad (3.7)$$

Intermolecular ligand-receptor scoring

ScalaVina, like AutoDock Vina, uses pre-computed 3D grid maps covering the volume of interest where the binding site is located. Each point in each grid contains the score an atom of type t would have if it were placed at exactly that location. Using interpolation, the scoring behaviour of the volume containing the binding site can be reasonably approximated, derivatives included. In order to nudge ligand atoms that may stray outside the volume, a penalty is awarded that is proportional to how far outside the grid the atom is, therefore guaranteeing that, eventually, all atoms will remain within the binding site volume (see figure 3.5).

ScalaVina takes advantage of Autodock Vina's grid map generation routine to create the initial grid map, which is then loaded to the GPU memory. Since the maps are the same on CPU and GPU, ligand-receptor score computing on CPU and GPU produce the same scores and derivatives. The code that evaluates an atom against the grid is functionally equivalent to the Autodock Vina code that does the same job, but the implementation has important differences.

In the GPU, the routine that does the computation runs all the energy calculations in parallel, one (or more) atom per thread, and accumulates the energy per thread until all atoms are done, at which point it performs a parallel reduction of all the separate energies into a single sum. Memory access patterns are not entirely favourable as atoms should be expected to fall on different grid map cells, but each memory access will usually load at least two necessary values to cache, one of which is immediately used. Furthermore, accessing the grid is minimized by only loading each grid point once and then computing all that can be computed from it. Coalesced memory access strategies cannot be employed on accessing the grid in this case, because the grid layout naturally prevents, in the common case, more than 2 of the 8 grid point values required to be loaded (per thread) at once. Restructuring the grid so that it is cell-based is an option to explore, but the overhead associated with the bookkeeping required to employ such a strategy, alongside the inevitable problem of the grid map using up to 8 times more memory, may raise the cost of the strategy high enough that it

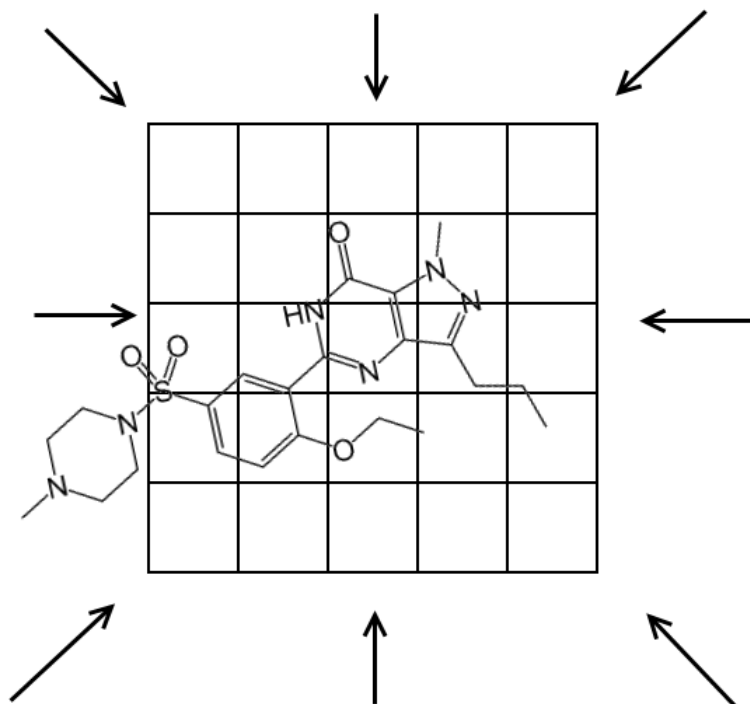


Figure 3.5 An example grid (in 2D) with a ligand partially inside. The arrows represent a force applied to the atoms that fall outside the box as a penalty

becomes a moot point. Coalesced access is therefore currently limited to the information of the atoms, like position, type and so on.

In the case of also computing the derivatives, they are stored in the atom forces' vector array, overwriting old values. Thus, computing intermolecular energies must always be done before intramolecular energies.

Intramolecular scoring

Like AutoDock Vina, intramolecular scoring in ScalaVina is based on a list of interacting pairs, where each element defines that a pair of atoms i and j interact with each other and need to be computed. This list of interactions is always much higher (except in trivial cases) than the number of atoms in the ligand, thus it is a problem that lends itself well to parallelism.

Interacting Pairs									
1,2	1,3	1,4	1,5	2,3	2,4	2,5	3,4	3,5	
0	1	2	3	4	5	6	7	8	
Table with hints (example with 6 columns)									
0	7					(at least 12 atoms needed to use most of the available space, packing algorithm also helps)			
1	5								
2	4								
3									
6									
8									

Figure 3.6 Illustration of a hint table built for an 8 interacting pairs set with a simple allocation algorithm. Utilization rate is low because there are too few atoms, a potentially common situation. The best form for the hint table to take, as well as the packing algorithm, is still uncertain.

ScalaVina, in the case where only the scoring is required, traverses in parallel the interacting pairs structure, one thread per interacting pair. Each thread starts by checking which atoms are involved and then load the ligand atom and position information for both i and j , then computes the score and locally accumulates it before carrying on to the next interacting pair. When all interacting pairs are evaluated, a simple parallel reduction accumulates all the separate energies into a single value.

Derivatives, however, are an entirely different problem. Derivatives are applied as forces in the forces' vector and, when done in parallel, may have more than one thread attempting to update the same vector at the same time. This data conflict is of critical importance towards efficiently solving the problem. ScalaVina deals with the problem by devising a method that guarantees that no two threads are simultaneously attempting to update the same vector, while exploiting as much parallelism as possible.

This is achieved with a hint table (see figure 3.6), where all the interacting pair indices are laid

out in rows of up to 32 elements, none of which interfere with each other. In other words, no two interacting pairs per row can have a common atom on either i or j . Such table enables ScalaVina to establish a secure order by which a single warp of 32 threads updates the atom's forces without any conflicts. However, because it is a relatively recent addition, not all options have been explored yet. The restrictions currently imposed are too strict to fill 32 threads when ligands are relatively small. In fact, any ligand smaller than 64 atoms cannot take full advantage of it. Relaxing the rules so that there can only not be two interacting pairs per row with the same atom as i or j (thus allowing, say, 0-1 and 1-2, which would be illegal under current rules) and carefully tuning the code so that it is guaranteed to not conflict, is likely to improve efficiency by a measurable amount.

Ligand force, torque and conformation update

The mechanical interpretation of derivatives in ScalaVina differ in no way from AutoDock Vina. However, the way in which force, torque and conformations are computed are very different. Because CUDA cannot deal with recursive functions, even if they were a practical option to parallelize the problem, a completely new solution had to be devised.

The tree-like structure, mapped into an array like mentioned before in section 2.2, can be navigated through iteration instead of recursion by computing and storing the depth of each node and iterating over what depth the algorithm may be at. This allows parallel traversal and computation of ligand structure data. Furthermore, full advantage of parallelism available is also taken by carefully wielding separate levels of parallelism as appropriate.

When computing the force and torque, the computation is broken into three separate steps. The first step uses a warp per node to compute the force and torque of the atoms belonging to that node in parallel, then accumulate the resulting values in the node data. The second step traverses the tree in a post-order fashion accumulating the force and torque of all the nodes until the final values are computed and stored at the root. The third step computes the derivatives of the position,

orientation and torsions from the computed forces and torque, one node per thread.

Conformation update traverses the tree in the other direction, one thread per node updating positions, orientations and axial information of the structural nodes only. Once that is done, all the atom's positions are updated in parallel, one thread per atom.

3.4 Search for global minimum

The Iterated Local Search algorithm implemented in ScalaVina is a close analogue to the one implemented in AutoDock Vina, but differs in a few ways. Unlike all of the remaining CUDA-based code for the molecular docking process, it is entirely sequential as there is nothing that can be parallelized.

The ILS algorithm implemented in the GPU code can be described as:

```

for  $n$  starting points do
   $S \leftarrow \text{GenerateInitialSolution}()$ 
  for  $step < \text{maxSteps}$  do
     $S' \leftarrow \text{Perturbation}(S)$ 
     $S'' \leftarrow \text{LocalSearch}(S')$ 
    if Metropolis accepts  $S''$  then
       $\hat{S} \leftarrow \text{append}(\hat{S}, S'')$ 
       $S \leftarrow S''$ 
    else
       $\hat{S} \leftarrow \text{append}(\hat{S}, S'')$ 
    end if
     $step \leftarrow step + 1$ 
  end for

```

end for

return \hat{S}

This variation of the ILS algorithm returns *all* minima and not just the ones accepted by the Metropolis criteria, but the criteria is still important because its acceptance determines from which state the next perturbation and local search will depart from. The task of filtering out duplicate minima is left to the CPU because it would be too expensive to sort and filter the results within the GPU. The filtering uses the functionality existent in AutoDock Vina to build the output conformations and rank them once the docking process in the GPU has ended.

The initial solution is a randomly generated conformation positioned within the search volume, but some of the ligand's branches may lie outside of it as the algorithm does not attempt to force all of its atoms to be within the search volume. Local search is performed only after the perturbation step in order to keep the algorithm code smaller, saving a local search call just after generating the initial solution. Perturbing an entirely random state in the first iteration should have no negative effect on the algorithm.

Parallelism at this stage can only be obtained by running concurrent ILS instances (represented by the outer loop in the algorithm) and by parallelizing the local search algorithm, which is precisely what has been achieved as described in their respective sections. However, since the algorithm is rather small and the individual sequential operations are fast, the impact on performance is minimal. This leaves the local search and all its related algorithms, such as scoring function and others, as the main resource hogs.

3.5 Search for local minima

The Broyden–Fletcher–Goldfarb–Shanno (BFGS) method is a numerical optimization method under the family of the Quasi-Newton methods. It is an iterative method that is of considerable

importance and utility. In ScalaVina, it is used to perform energy minimizations using the scoring function. Here, the used BFGS method is broken down into its constituent parts and explained, showing data dependencies and possible optimization paths. For this reason, a more theoretical analysis of the algorithm and its components is performed.

Algorithm Structure

The BFGS method is an iterative algorithm approximating to the Hessian matrix of a function $f(x)$.

The algorithm can be described as:

1. Initialize $B_0^{-1} = I$, $k = 0$, x_0 to an initial guess and ε to a really small value (example 10^{-4}).
2. Initialize $g_k = \nabla f(x_0)$
3. End if $|\max g_k| < \varepsilon$
4. Solve $p_k = B_k^{-1}(-g_k)$
5. Perform a line search using p_k and x_k , obtaining an appropriate α .
6. Solve $s_k = \alpha p_k$, then $x_{k+1} = x_k + s_k$ and $y_k = \nabla f(x_{k+1}) - g_k$
7. Solve:

$$B_{k+1}^{-1} = B_k^{-1} + \frac{(s_k^T y_k + y_k^T B_k^{-1} y_k)(s_k s_k^T)}{(s_k^T y_k)^2} - \frac{B_k^{-1} y_k s_k^T + s_k y_k^T B_k^{-1}}{s_k^T y_k} \quad (3.8)$$

8. Set $x_k = x_{k+1}$, $g_k = \nabla f(x_k)$, $B_k^{-1} = B_{k+1}^{-1}$, $k = k + 1$ then jump back to step 3

This algorithm deserves some comments. The first is that B_k^{-1} does refer to the inverse of a matrix B_k that is not being calculated. This is because it is not necessary to do so as B_k is not used

at any stage. If there was such a need, step 4 would require solving $B_k p_k = g_k$, a calculation that is ordinarily done using the inverse of B_k anyway. By iteratively updating the inverse of B_k instead of generating it from B_k every iteration, the algorithm saves enormous amounts of calculations. For reference, the equation that updates B_k is as follows:

$$B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T s_k} - \frac{B_k s_k (B_k s_k)^T}{s_k^T B_k s_k} \quad (3.9)$$

The second observation is that $s_k^T y_k$ is calculated multiple times, so it is only normal that it should be calculated only once and reused in implementations to avoid redundant work. This may not make sense in computing architectures where memory is at a premium, so it needs to be evaluated on a case by case basis but, considering the result is a scalar value, it is extremely unlikely not to be advantageous to do so. A third observation is that the line search algorithm is responsible for optimizing how far in the direction p_k the iteration goes to make convergence faster.

Finally, the function $f(x)$ itself cannot be picked randomly. It must respect certain properties in order to guarantee the BFGS method makes sense and converges to a local minima.

Line search

Line search methods are designed to answer the question of how far in direction p_k we wish to go, and are a critical component of any gradient-based search algorithm. The way ScalaVina performs a line search is to use a backtracking line search like this:

1. Let p_k be the descent direction to be searched and x_k the position from where p_k departs
2. Set $l = 1$ as an iteration counter, c_0 as a factor for the Armijo condition and $\tau \in]0, 1[$ as the rate at which α is decreased per iteration
3. Set $f_0 = f(x_k)$, $\alpha_l = 1$

4. Solve $x_{new} = x_k + \alpha_l p_k$ and $f_l = f(x_{new})$
5. If $(f_l - f_0) < \alpha_l c_0 p_k^T \nabla f(x_k)$, α_l satisfies the Armijo condition and the line search is over
6. Else, set $\alpha_{l+1} = \tau \alpha_l$, $l = l + 1$ and go back to step 4

Additional or different conditions may be imposed on step 5, like the Wolfe conditions used in later line search algorithms [9]. The Armijo condition alone guarantees that α is not too large, so for most purposes it should be enough. In addition, the Wolfe conditions require the computation of $\nabla f(x_{new})$ in every iteration, which is a relevant factor to consider.

Scoring function $f(x)$ requirements

The BFGS algorithm requires a function f with specific requirements in order to operate. Therefore, any function f should be such that

$$\min f(x) \text{ subject to } x \in \mathbb{R}^n \quad (3.10)$$

In order for (3.10) to be meaningful, f must be bounded from below. However, while this guarantees a lower bound, it does not guarantee an optimal solution. A function f that is guaranteed to have a global optimum x^* must be continuous within a bounded set. [9]

Now that it is established that a global optimum must exist, it is necessary to determine how optimum points can be recognized. There are necessary and sufficient conditions that are well known, as below [9]:

- Necessary conditions: if x^* is optimal, then
 - 1st-order necessary condition: the gradient $f'(x^*)$ is zero;
 - 2nd-order necessary condition: the Hessian $f''(x^*)$ is positive semi-definite.

- Sufficient condition: if x^* is such that $f'(x^*) = 0$ and $f''(x^*)$ is positive definite, then x^* is a local minimum (i.e. $f(x) \geq f(x^*)$ for x close to x^*).

Therefore, it is natural to conclude that any function f would need to have a lower bound and at least be continuous across the (relevant) domain.

Implementation

The CUDA implementation of the BFGS algorithm is built as a C++ template function so that scoring function and some other parameters may be freely swapped on compile time. It extensively uses helper functions to implement the algebraic operations and stays true to the algorithm presented. This is, however, not optimal. The presented, and implemented, algorithm is a reference algorithm and has a lot of intermediate computations that are stored in intermediate structures, increasing memory usage. Unfortunately, it was not possible to get around to fine tuning the BFGS algorithm beyond the helper functions, which all work in parallel and have either optimal or reasonable implementations. Intermediate data structure elimination should provide a relevant performance boost even if it may come at the cost of duplicating computations.

An alternative BFGS has also been implemented, mirroring the code in AutoDock Vina but with as much parallelism as possible. This one has been confirmed to be faster than the reference algorithm, but for some reason produces lower quality results. The reason is unclear yet, as the code is comparatively opaque. Since there is also no documentation detailing AutoDock Vina's exact implementation of the BFGS algorithm, it was not a priority to investigate the reason.

Improving the BFGS algorithm is a priority for the future, and the most likely useful path towards any further improvement is eliminating as much intermediate computation storage as possible.

Chapter 4

Validation and Assessment

In this chapter the preliminary results of the tests performed in ScalaVina are shown and analysed. Two different versions of the ScalaVina code are considered, the first production reference code which provided full functionality and a subsequent version with some additional optimization work. The version of AutoDock Vina considered is version 1.1.1 as available from its website without modifications.

4.1 Functional validation

Validating the functionality of ScalaVina was performed at all stages of development, directly comparing whenever possible the output of the GPU code with the output of the equivalent CPU code. All code performed within expected parameters, though slight deviations in some computations (in the order of 10^{-15}) were accepted and the scoring function is known to not match the pre-calculated look-up table present in AutoDock Vina.

Validation of the entire molecular docking process is a more complex undertaking because even minor differences in the local search algorithm and parameters will invalidate direct comparisons. Therefore, the validation takes into account mainly whether it finds the best conformations from

both the free binding energy predicted perspective and the actual conformation predicted, namely how close it comes to the binding site and what shape it takes in it compared to the experimentally derived conformation. A measurement technique called root mean square deviation (RMSD) is used to assess how different the two conformations are by determining the average distance between the atoms' positions. The smaller the RMSD between the computed conformation the better.

To validate the accuracy of the ScalaVina implementation, we performed a series of docking experiments using a subset of 24 selected protein-ligand complexes from a larger dataset that is used to validate docking algorithms and programs [16]. In this set, the receptors are considered rigid, and the ligands are treated as flexible molecules with the number of active rotatable bonds ranging from 2 to 16. In the docking calculations, all possible torsions of the compounds were set flexible and the target rigid. The grid for probe-target energy calculations and conformational searching was placed with its centre at the enzyme-binding site. The docking grid size was, at most, $40 \times 40 \times 40$ grid points with 1.0 Å spacing. The resulting docking solutions were ranked accordingly to their binding free energy, and lowest one was selected.

In figure 4.1 the ligand RMSD of each docking experiment as computed in ScalaVina and AutoDock Vina are shown. Both follow the same trends and both are good (or bad) in the same docking experiments, showing that ScalaVina and Vina are equivalent. Table 4.1 shows that, in fact, the fluctuations average out without giving either a clear advantage. This may allow for overall better results if the execution parameters are tweaked and both a CPU and a GPU simulation's outputs are combined. How that may be done exactly will have to be determined at a later time, but it presents a clear opportunity to improve result quality beyond what is already possible. The table indicates that the average of the best results obtained in the validation runs from both ScalaVina and AutoDock Vina against the experimental results average out at an RMSD 0,18 Å closer to the experimental results than either of them alone. It is relevant to keep in mind that the RMSD

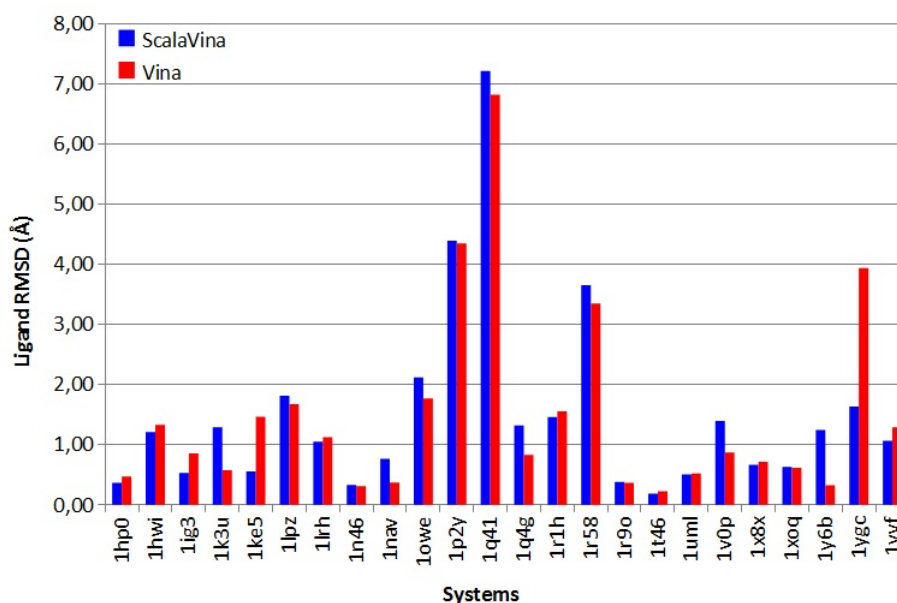


Figure 4.1 ScalaVina and Vina ligand RMSD (Å) from experimental ligand structure. Also see table 4.1

differences observed between Vina and ScalaVina are small and acceptable for this type of quality (see figure 4.2).

4.2 Testbed platform

All tests were executed in a pair of nodes installed in the SeARCH cluster at the Universidade do Minho. The main characteristics of these nodes are:

- 2 AMD CPUs with 12 cores each (AMD Opteron 6174, 2.2GHz, 12MB L3 cache)
- 2 NVidia Tesla C2050 cards (Fermi architecture, 448 cores, 3GB RAM)
- 64GB ECC RAM
- Linux with CUDA 4.0, GCC 4.1.1.2, Boost 1.47

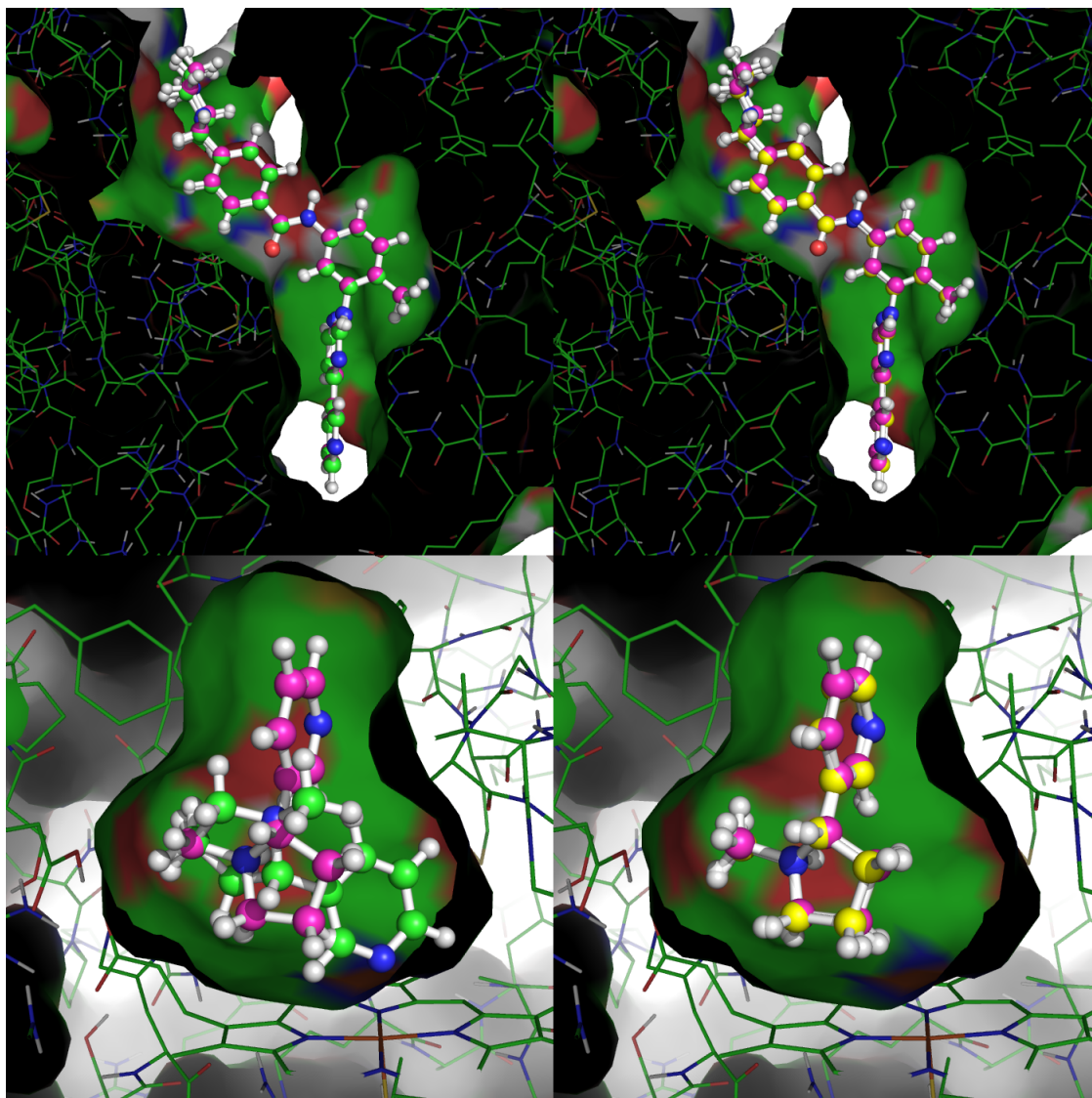


Figure 4.2 Protein:ligand structural complexes of the best (upper panels) and worst (lower panels) docking predictions by ScalaVina and Vina. Protein targets are rendered in lines and a molecular surface with the following color code: Carbons—green, Nitrogen—blue, Oxygens—red, Hydrogen—white. Ligands are rendered in a ball-and-stick style with different colouring codes for the carbon atoms. Molecular surface is slanted in order to visualize the interior of the target’s binding pocket. Experimental ligand conformations are colored with carbons in green, ScalaVina ligand conformational predictions with carbons in magenta and Vina ligand conformational predictions with carbons in yellow. Upper left panel: Comparison of the ScalaVina ligand conformation prediction against the experimentally known result for model 1t46. Upper right panel: Comparison between the ScalaVina and Vina ligand conformation predictions for model 1t46. Lower left panel: Comparison of the ScalaVina ligand conformation prediction against the experimentally known result for model 1q41. Lower right panel: Comparison between the ScalaVina and Vina ligand conformation predictions for model 1q41 (See table 4.1 for RMSD values).

Table 4.1 RMSD (Å) from experimental ligand structural data

Model	Vina	ScalaVina	Best
1hp0	0,47	0,36	0,36
1hwi	1,33	1,21	1,21
lig3	0,85	0,53	0,53
1k3u	0,57	1,28	0,57
1ke5	1,46	0,55	0,55
1lpz	1,67	1,81	1,67
1lrh	1,12	1,05	1,05
1n46	0,31	0,33	0,31
1nav	0,37	0,76	0,37
1owe	1,77	2,12	1,77
1p2y	4,35	4,39	4,35
1q4l	6,82	7,21	6,82
1q4g	0,83	1,32	0,83
1r1h	1,55	1,46	1,46
1r58	3,35	3,65	3,35
1r9o	0,36	0,38	0,36
1t46	0,22	0,18	0,18
1uml	0,52	0,50	0,50
1v0p	0,87	1,39	0,87
1x8x	0,71	0,66	0,66
1xoq	0,61	0,62	0,61
1y6b	0,32	1,24	0,32
1ygc	3,94	1,63	1,63
1yvf	1,29	1,06	1,06
Average	1,49	1,49	1,31

Despite the presence of two Tesla cards, ScalaVina is only currently using one so all docking simulations were executed in only one GPU. AutoDock Vina was tested as available for download and not recompiled.

4.3 Performance evaluation

Execution times of molecular docking simulation is a critical issue. The current prototype version of ScalaVina has not been seriously tuned yet, presenting performance results below the initially expected values as can be seen in figure 4.3 and table 4.2. Despite the improvements in the kernel of version 2 (most notably the implementation of the hint table described earlier), the speed-up is still not enough to make it competitive against the highly tuned AutoDock Vina when it starts using multiple cores. However, it has also become clear since that there are important bottlenecks currently limiting the GPU code from achieving its potential.

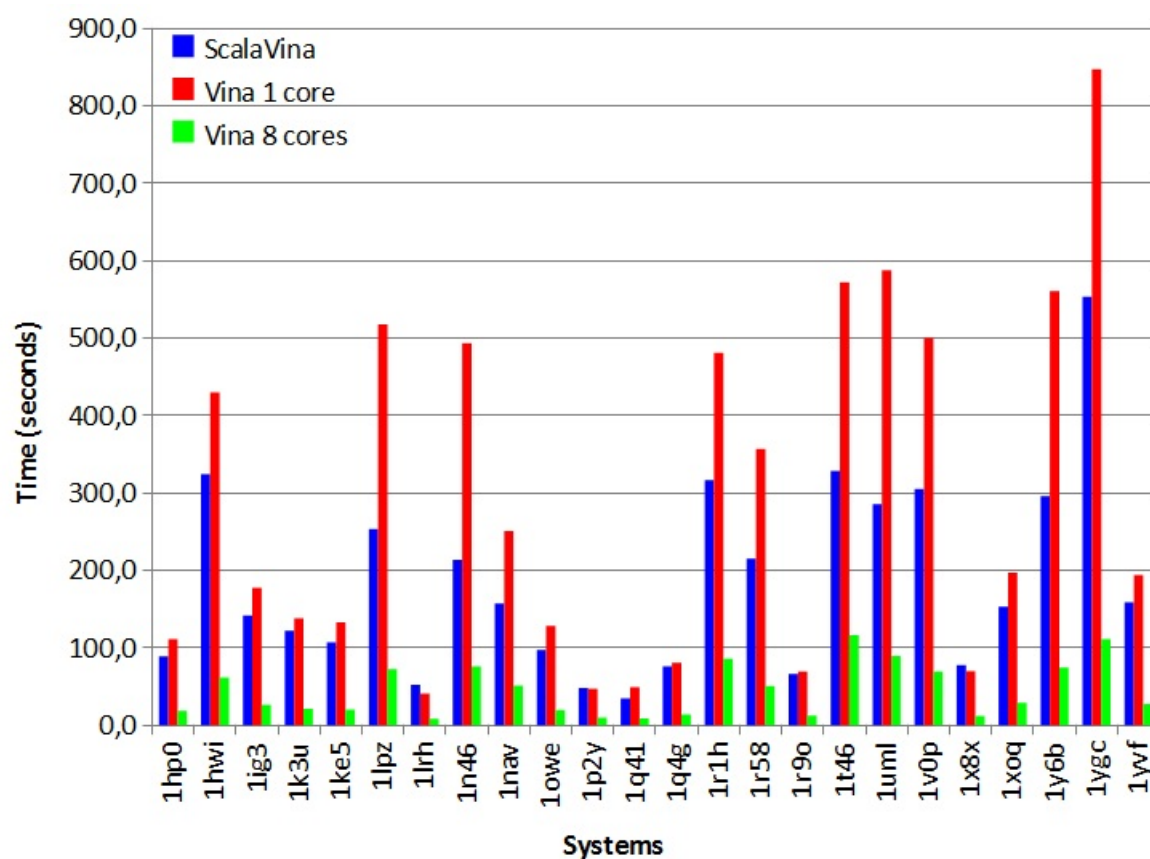


Figure 4.3 Runtimes in seconds for ScalaVina and AutoDock Vina for a set of systems at default configuration

Table 4.2 Runtimes in seconds for ScalaVina and AutoDock Vina for a set of systems at default configuration

Model	DOF	Vina 1 core	Vina 4 cores	Vina 8 cores	ScalaVina, v1	ScalaVina, v2
1hp0	11	110,7	30,5	18,6	191,4	88,9
1hwi	18	429,4	112,7	61,4	621,1	323,9
1ig3	14	177,1	46,4	26,1	291,8	141,4
1k3u	12	137,6	36,5	21,0	244,0	121,8
1ke5	10	132,5	34,4	20,1	210,0	106,7
1lpz	14	517,8	135,2	72,0	527,4	253,4
1lrh	8	40,6	11,6	8,2	99,5	52,0
1n46	15	493,5	145,3	75,8	419,0	213,2
1nav	14	250,7	76,2	50,9	312,3	156,8
1owe	10	128,0	34,3	19,3	197,3	96,8
1p2y	8	46,9	13,4	9,4	100,3	47,7
1q41	7	48,7	14,2	8,8	63,0	34,3
1q4g	10	80,5	21,8	13,7	157,1	75,6
1r1h	18	481,0	144,9	85,5	651,5	316,5
1r58	18	356,6	3,7	50,3	426,1	214,8
1r9o	9	69,2	18,9	12,1	135,3	66,0
1t46	14	571,6	194,3	116,3	659,3	328,1
1uml	17	587,4	155,8	89,2	583,6	285,2
1v0p	18	500,0	129,3	68,9	623,1	304,9
1x8x	11	69,7	19,1	11,9	156,0	77,1
1xoq	13	196,8	51,6	28,7	299,8	152,8
1y6b	15	560,4	143,3	74,5	617,8	295,8
1ygc	22	847,3	214,6	111,1	1118,8	553,3
1yvf	12	193,7	50,0	27,4	317,4	158,4
average		292,8	76,6	45,0	376,0	186,1

A third version currently in development and not shown here shaves off about a third of the runtime in a reference test by limiting the size of the ligand it can handle both in number of atoms (down to 80) and degrees of freedom (down to 25) and storing all the ligand's atoms data in shared memory. An attempt to move the interacting pairs into shared memory, however, proved impossible because it requires too much memory. Furthermore, it is now clear that the interacting pairs structure is the most important bottleneck to defeat at the moment, since it is where the largest share of the workload of the intramolecular part of the scoring function is at, and it is always used every time the scoring function is called.

The next planned version will develop a radically modified `cudaModel` structure to reduce its memory usage, most notably by downsizing almost every 32-bit integer into an 8-bit integer, potentially downsize atom position coordinates from double-precision FP values to single-precision FP and restructure the interacting pairs data structure into using 8 bits per atom index as well. While such an approach would limit indexing to 256 atoms, it would not actually be a problem because it is not likely even half that many atoms could be stored in shared memory. Also, multiple kernel variants can potentially be provided with different optimizations and limitations should it prove to be possible and beneficial, with an unlimited, slower version always available as a fall back.

Other routes of optimization include developing an additional scoring function transformation to smooth it, similar to how Vina smooths the pre-calculated scoring function look-up table. Despite the almost certain higher cost of computing the scoring function that adding such an operation would have, the current situation is almost certainly worse. It has been observed that, while Vina computes on average in the reference test 3 line searches per BFGS iteration, ScalaVina has been observed to compute almost 7, effectively doubling the computing work it must perform in a docking experiment as compared to Vina. Lowering the number of line searches performed per BFGS iteration would be of great help towards faster docking.

Chapter 5

Conclusions

A molecular docking prototype capable of running simulations on both CPU and CUDA-enabled devices was developed. The ScalaVina prototype is fully functional for single ligand:receptor molecular docking and evidences close affinity with the results Vina produces. ScalaVina is, however, not yet fine tuned and does not provide an advantage over multi-core CPUs in simulation runtime. The main bottlenecks have been identified and are being tackled one at a time, and it is expected that eventually ScalaVina's CUDA simulation code will surpass the multi-core CPU simulation code in speed.

Thoughts and future work

When I started working on ScalaVina, I only had ideas of how molecular docking worked and of how Vina worked internally. Studying Vina's source code yielded a significative amount of information, but it was not until I started implementing ScalaVina that the breadth started becoming clear.

Among the early tasks performed in preparation for coding was a study of the BFGS algorithm, at the time thought to be probably where I would have to focus the most attention on. The theoret-

ical and practical analysis, including implementing it in Octave, was instrumental in building the reference implementation. Back then I wrote most of a technical report on it, which I reused many parts of here. It is a practice I now recommend anyone writing a thesis to do, for time well spent early on can save a lot of time later when crunch time comes.

Working out early implementations of scoring took some time but for the most part there were no surprises there. A lot of time was spent building up the base infrastructure upon which ScalaVina stands, making sure the data structures were being converted to their CUDA equivalents correctly and data movement routines to and from the GPU were working. I could afford no errors in any of that and the verification routines written greatly helped in that regard.

Once I had the foundation to build up on, the scoring function was relatively easy to implement. Intramolecular energy was almost trivially easy to write a working implementation of (but no derivatives yet then) and intermolecular settled in after I chose to use Vina's grids as a base and fixed some stealthy bugs that crept in on the code produced. It was when I turned to conformations that I realised there was a lot more that looking over code had not made clear. Completely reworking how conformations were computed for a parallel architecture was a challenge but a satisfying one, especially as it works so well.

I then moved to the BFGS algorithm and, of course, the derivatives. By far, this was the most difficult part. The BFGS algorithm kept failing to work correctly, the derivatives required more deep algorithm reworking (fortunately sped up by being fresh out of implementing the conformation update code) and unbeknownst to me it later would turn out the implementation I had for intramolecular derivative computation was flawed. After debugging the BFGS algorithm itself and fixing the intramolecular derivatives, I could finally see it all coming together, although I didn't know yet that the high performance I was expecting wasn't going to materialize so easily.

The ILS algorithm did not take long to put together, after some technical issues were worked out. Then the first full tests started and it became clear that there were problems. One fix for the

interactive pairs derivatives (in the form of a hint table) was implemented which, along with some fine tuning on shared memory usage, reduced runtime length by about a factor of two. It was still too slow, though.

It is perhaps ironic that the most valuable lesson was learned by seeing the docking simulations take longer than I had imagined it would, but I learned a lot about the *entire* algorithm in the process. Some of my initial assumptions were faulty, as I had expected to have good performance by relying on the cache for most of the work and judiciously using shared memory to speed up the BFGS algorithm, reductions and other opportunity targets. The actual size of the interacting pairs structure, along with other unforeseen factors, makes that plan unlikely to work.

Future work will center on creating limited kernels that put as much data in shared memory as possible while avoiding performance cliffs. A current kernel already does that with ligand atomic information and structural information, to a gain of around 30% speed and without a doubt there is more gain to be had. Once a clearly acceptable level of performance at this level has been achieved, the BFGS and line search algorithms will be looked at for improvement. Looking into how to implement residues and flexible side-chains is also planned, though it will have to wait for now. Eventually, kernels capable of using shared memory to the fullest while not being limited in the problem size would be the ideal outcome.

Bibliography

- [1] D.R.Flower (Editor), *Drug Design: Cutting Edge Approaches* (Royal Society of Chemistry, Cambridge, 2003).
- [2] T. Fink, B. H, and R. J.-L, “Virtual Exploration of the Small-Molecule Chemical Universe below 160 Daltons,” *Angewandte Chemie International Edition* **44**, 1504–1508 (2005).
- [3] R. Wang, X. Fang, Y. Lu, C.-Y. Yang, and S. Wang, “The PDBbind Database: Methodologies and Updates,” *J. of Med. Chem.* **48**, 4111–4119 (2005), pMID: 15943484.
- [4] M. Taufer, M. Crowley, D. Price, A. Chien, and C. Brooks III, “Study of a Highly Accurate and Fast Protein-Ligand Docking Algorithm Based on Molecular Dynamics,” *Parallel and Distributed Processing Symposium, International* **10**, 188 (2004).
- [5] O. Trott and A. Olson, “AutoDock Vina: Improving the Speed and Accuracy of Docking with a New Scoring Function, Efficient Optimization, and Multithreading,” *J. of Comp. Chem.* **31**, 455–461 (2010).
- [6] S. Huang, S. Grinter, and X. Zou, “Scoring functions and their evaluation methods for protein-ligand docking: recent advances and future directions,” *Phys. Chem. Chem. Phys.* **12**, 12899–12908 (2010).

-
- [7] R. Abagyan, M. Totrov, and D. Kuznetsov, "ICM - A new method for protein modeling and design: Applications to docking and structure prediction from the distorted native conformation," *J. of Comp. Chem.* **15**, 488–506 (1994).
- [8] J. Nocedal and S. Wright, *Numerical Optimization, Springer Series in Operations Research*, 1st ed. (Springer-Verlag, Heidelberg, Berlin, 1999).
- [9] J. F. B. et al, *Numerical Optimization: Theoretical and Practical Aspects*, 1st ed. (Springer-Verlag, Heidelberg, Berlin, 2003).
- [10] G. Morris, D. Goodsell, R. Halliday, R. Huey, W. Hart, R. Belew, and A. Olson, "Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function," *J. of Comp. Chem.* **19**, 1639–1662 (1998).
- [11] N. Vanopdenbosch, R. Cramer, and F. F. Giarrusso, "SYBYL, the integrated molecular modeling system," *J. Mol. Graph* **3** .
- [12] R. Wang, L. Lai, and S. Wang, "Further development and validation of empirical scoring functions for structure-based binding affinity prediction," *J. of Comp-aided Mol. Des.* **16**, 11–26 (2002).
- [13] A. N. Jain, "Scoring noncovalent protein-ligand interactions: a continuous differentiable function tuned to compute binding affinities.," *J. of Comp-Aided Mol. Des.* **10**, 427–440 (1996).
- [14] D. B. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach* (Morgan Kaufmann, 2010).
- [15] NVIDIA, "Fermi Compute Architecture White Paper," .

-
- [16] M. J. Hartshorn, M. L. Verdonk, G. Chessari, S. C. Brewerton, W. T. M. Mooij, P. N. Mortenson, and C. W. Murray, “Diverse, High-Quality Test Set for the Validation of Protein-Ligand Docking Performance,” *J. of Med. Chem.* **50**, 726–741 (2007), pMID: 17300160.

